Final: Incompressible, Laminar Flow over a Rectangular Cavity

John Karasinski
Graduate Student Researcher
Center for Human/Robotics/Vehicle Integration and Performance
Department of Mechanical and Aerospace Engineering
University of California
Davis, California 95616
Email: karasinski@ucdavis.edu

1 Problem Description

Forty five years ago, Mehta and Lavan published a paper on the numerical investigation of flow over a rectangular cavity at low Reynolds numbers [1]. This relatively simple geometry provides tremendous insight into the physics of flow separation, an important flow feature in many applications. A numerical 2D planar model of these incompressible, laminar flows is developed here. In particular, the predicted flow structure (streamline pattern, eddies) and velocity profiles are investigated for a variety of aspect ratios (AR) and Reynolds numbers (Re).

2 Numerical Solution Approach

The icoFoam solver from OpenFOAM 2.3.0 was used to computationally solve the Navier-Stokes equations,

$$\rho\left(\frac{\partial \mathbf{v}}{\partial t} + \mathbf{v}\cdot\nabla\mathbf{v}\right) = -\nabla p + \mu\nabla^2\mathbf{v} + \mathbf{f}. \quad (1)$$

icoFoam is a transient solver for incompressible, laminar flow of Newtonian fluid. The geometry for this problem consists of a channel of width 10 m and height 1 m. A cavity is placed just below the center of this channel and has a width of 1 m and a depth of AR (see Figure 1). Five cases were investigated: a base case with AR=0.5 and Re=100, and additional cases of AR=0.5 with Re=1 and 2000 for AR=0.5, and AR=2.0 and 5.0 for Re=100. A Python script was created to generate the initial conditions, geometry, and meshes for each case.

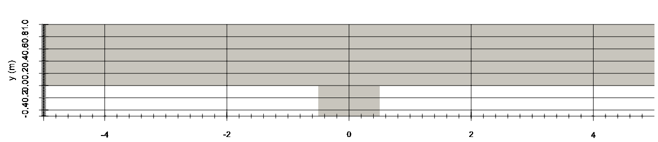


Fig. 1: Geometry for the AR=0.5 cases (width of channel = 10m, height of channel = 1m; width of cavity = 1m, height of channel = 0.5m)

2.1 Initial and Boundary Conditions

The initial velocity field is set to zero throughout the domain. Additionally, the top lid of the channel is set to move to the right at 1 m/s and the velocity at the walls is set to zero. The initial pressure field is also set to zero throughout the domain, and the lid and walls are set to a “zeroGradient” condition. The boundary field for the inlet and outlet boundaries are set to “zeroGradient” for all of the initial fields, and the boundary field for frontAndBack is set to “empty” for all initial fields, turning this into a 2D problem.

2.2 Geometry and Mesh

The script also generated the nonuniform mesh for this geometry using the `blockMesh` tool. This mesh was divided into four blocks: the left half of the channel, the right half of the channel, the center of the channel, and the cavity. For the base case, both the left and right halves of the channel were split into 100x100 points in the x, y direction, and also used “simpleGrading” in the x-direction to grade the meshes to be denser towards the center of the domain. The center

Name	x	y	simpleGrading
Left channel	M	M	(5/M 1 1)
Right channel	M	M	(M/5 1 1)
Central channel	M	M	(1 1 1)
Cavity	M	M*AR	(1 1 1)

Table 1: Mesh configuration algorithm (M=100, AR=0.5 for base case)

of the channel was also split into 100x100 points in the x, y direction. The cavity for the base case was split into 100x50 points in the x, y direction. A generalized version of this mesh is available in Table 1 (where M is some desired mesh size).

The resulting mesh was then split on to four CPUs using the `decomposePar` tool, and the icoFoam solver was called with the MPI option. The solver than solved the system to the specified end time and reconstructed the domain and fields using the `reconstructPar` tool. Solving to a solution time of 10 seconds for the base case takes 92.82s on a single core and only 44.01s on four cores, leading to a speedup of 2.1. This speedup could likely be improved by changing the distribution of the mesh onto the processors.

3 Results Discussion

3.1 Comparison to Published Results

Constant streamline contours were plotted and compared to previously published results by Mehta and Lavan [1]. The streamline contours for AR=0.5 with Reynolds numbers of 1 and 100 are shown in Figures 4 and 5. They key characteristic of these cases is the concase nature of the upper streamline in the Re=1 case, and the convex nature of the upper streamline in the Re=100 case. Both of these cases compare well to published results.

The case of AR=2, Re=100 also compares well with the published results. The same convex nature found in the AR=0.5, Re=100 is also present here, as is the additional formation of the lower vortex. Upon closer inspection, two corner eddies are apparent in the upstream and downstream corners (see Figure 7). The published results suggest the formation of an additional, smaller corner eddy in each corner. This result is not seen here, though a mesh refinement would likely make these apparent.

3.2 Additional Cases

Streamline contours were also found for AR=5, Re=100. This case shows the same top vortex as in the AR=2, Re=100 case, but also shows the the formation of three additional vortices. The upper two of these vortices appear to be the same, while the bottom vortex appears to be very similar to the bottom vortex in the AR=2, Re=100 case. These results can be seen in Figure 8.

The case of the AR=0.5, Re=100 case was also investigated. This streamline contours for this case are similar to that for the AR=0.5, Re=100 case, but bulge out of the cavity on the right side.

3.3 Velocity Profile Convergence

The system was considered converged when the velocity profile before and after the cavity both approximated a linear function. This is accomplished by sampling the velocity profile at x = 1 m and x = -1 m with the `sample` tool and plotting against y. An example of this velocity profile for the base case at $t = 30$s can be seen in Figure 2.

The NRMS as a function of time is also investigated. A linear velocity profile was taken as the exact solution, and was used as the basis for comparison between the computational solutions found at each time step. The Root Mean Square error,

$$RMSE = \sqrt{\frac{1}{N}\sum_{i=1}^{N}[U_i - U^*]^2}, \tag{2}$$

and the Normalized Root Mean Square error,

$$NRMS = \frac{RMSE}{max(U^*) - min(U^*)}, \tag{3}$$

can be calculated (as $max(U^*) - min(U^*) = 1$ in this case, the $NRMS = RMS$). Here U_i is the computational result at each sampled point, U^* is the linear velocity profile solution, and N is the number of sampled points. The $NRMS$ for each case is expressed as a percentage, where lower values indicate a result closer to the analytic solution.

The results of NRMS of the base case at each time step for the velocity profiles before and after the cavity are plotted in Figure 3. This shows that the convergence of the velocity profile after the cavity converges to $NRMS \approx 2E-4$ at approximately $t = 120$s, while the velocity profile before the cavity continues to decrease logarithmically. The convergence of the velocity profile after the cavity suggests that the vortices in the cavity are also effectively converged by this time.

4 Conclusion

References

[1] Mehta, U. B., and Lavan, Z., 1969. "Flow in a two-dimensional channel with a rectangular cavity". *Journal of Applied Mechanics,* **36**(4), pp. 897–901.

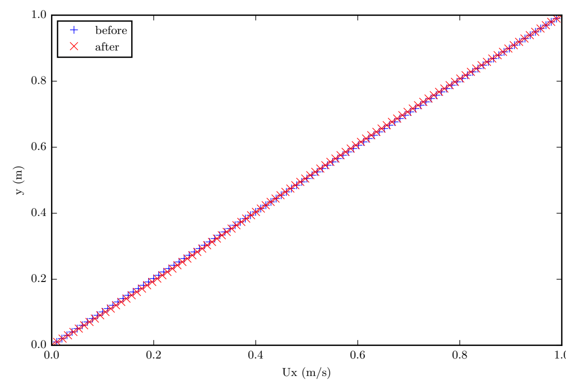


Fig. 2: Velocity profile of Ux before ($x = 1$m) and after ($x = -1$m) cavity, for base case at $t = 30$s

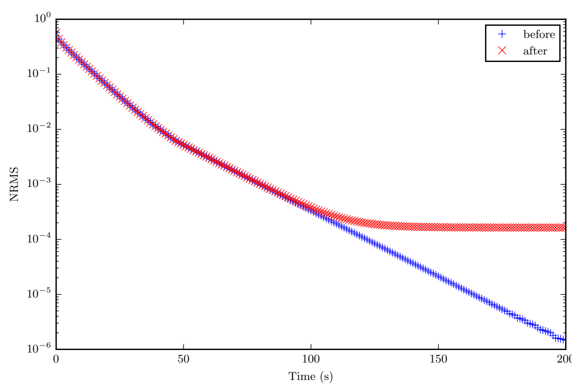


Fig. 3: Convergence of velocity profile before ($x = 1$m) and after ($x = -1$m) cavity, for base case

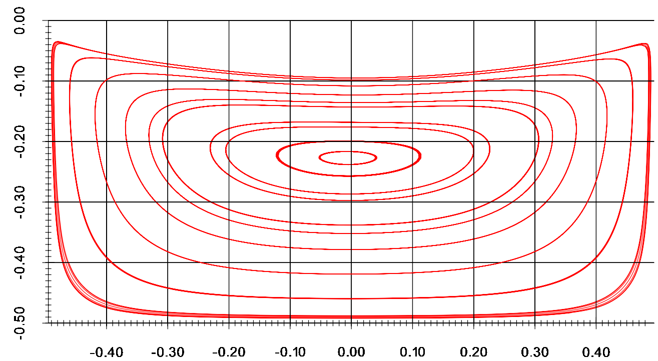


Fig. 4: AR=0.5, Re=1

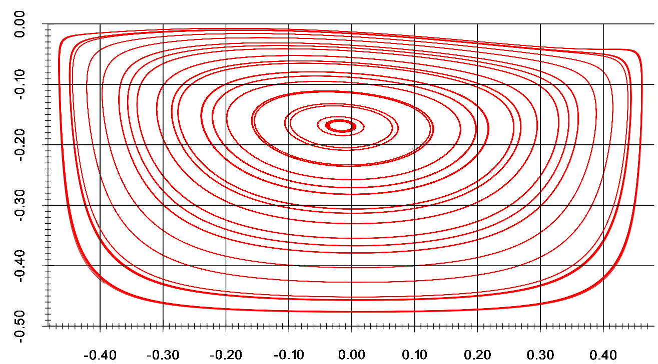


Fig. 5: AR=0.5, Re=100 (base case)

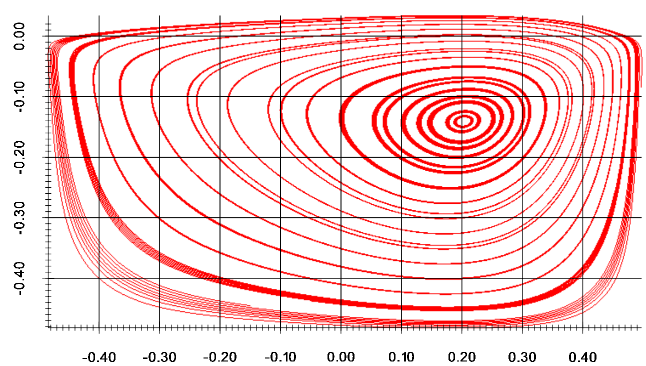
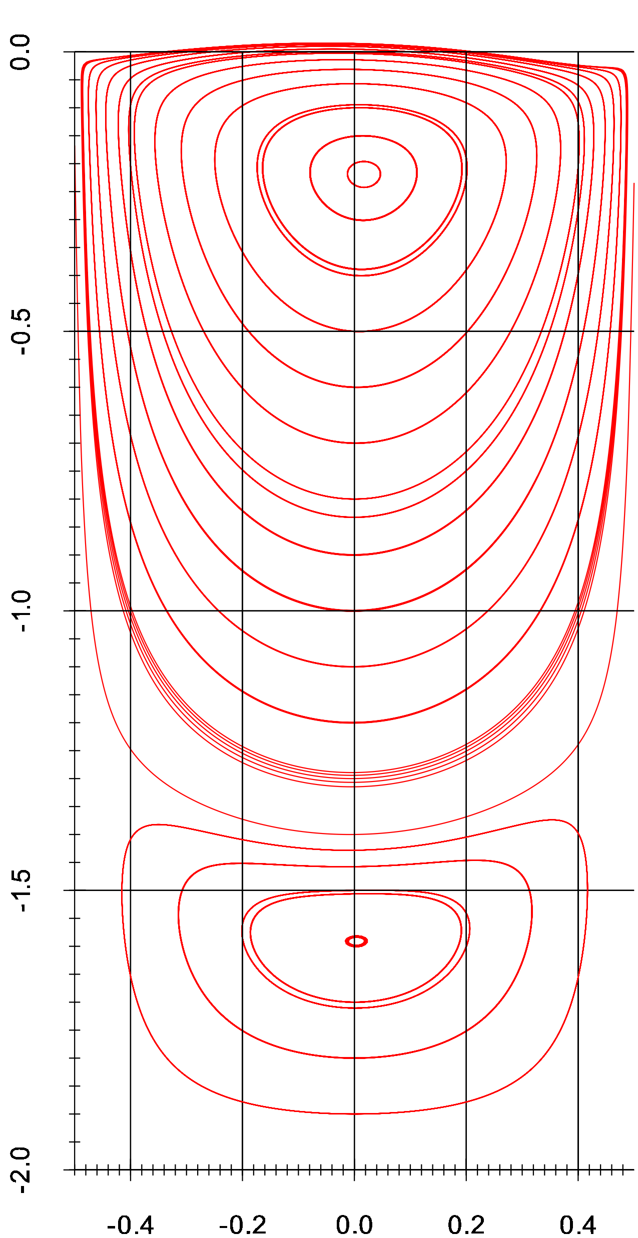
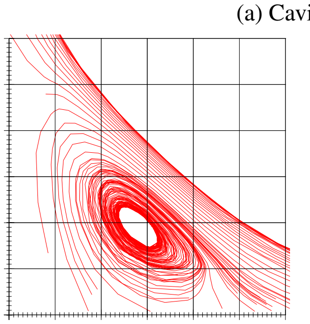


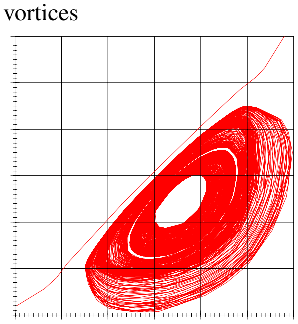
Fig. 6: AR=0.5, Re=2000



(a) Cavity vortices



(b) Upstream corner eddy, large ticks are 0.01m ($y \in \{-1.94, -2.00\}$, $x \in \{-.50, -.44\}$)



(c) Downstream corner eddy, large ticks are 0.01m ($y \in \{-1.94, -2.00\}$, $x \in \{.44, .50\}$)

Fig. 7: AR=2, Re=100

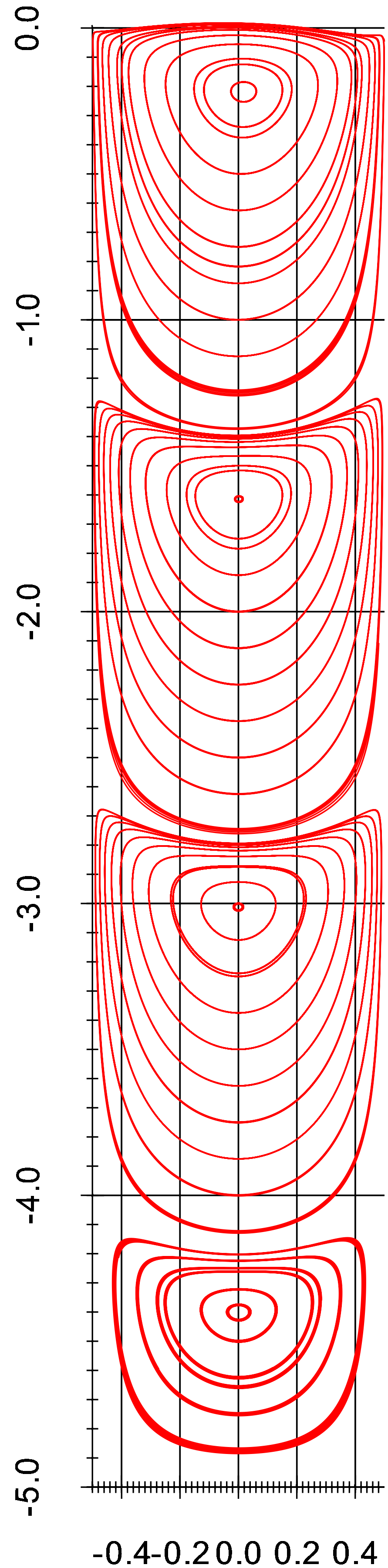


Fig. 8: AR=5, Re=100

Appendix A: Python Code

```python
import subprocess
import os


def inplace_change(filename, old_string, new_string):
    with open(filename) as f:
        s = f.read()

    if old_string in s:
        # print 'Changing "{old_string}" to "{new_string}"'.format(**locals())
        s = s.replace(old_string, new_string)
        with open(filename, 'w') as f:
            f.write(s)
    # else:
        # print 'No occurances of "{old_string}" found.'.format(**locals())


def subprocess_cmd(command):
    process = subprocess.Popen(command, stdout=subprocess.PIPE, shell=True)
    proc_stdout = process.communicate()[0].strip()
    # print proc_stdout


def generate_folders(ARs, Res):
    for AR, Re in zip(ARs, Res):
        run = "Run" + str(AR) + '-' + str(Re)
        if not os.path.exists(run):
            command = "cp -rf base/ " + run + "/; "
            subprocess_cmd(command)

    print ('Folders generated.')


def create_mesh_file(path, AR, Re):
    M = 1000.

    YMESH       = str(int(M * AR))
    INV_GRADING = str(0.1)
    GRADING     = str(10.)
    MESH        = str(int(M))
    AR          = str(-AR)

    inplace_change(path, 'AR',                   AR)
    inplace_change(path, 'XMESH',              MESH)
    inplace_change(path, 'YMESH',             YMESH)
    inplace_change(path, 'INV_GRADING', INV_GRADING)
    inplace_change(path, 'GRADING',         GRADING)
    inplace_change(path, 'MESH',               MESH)


def create_properties_files(path, AR, Re):
    d = 1.           # characteristic size of domain
    NU = str(d / Re)

    inplace_change(path, 'NU_VAR', NU)


def update_dimensions(ARs, Res):
    for AR, Re in zip(ARs, Res):
        run = "Run" + str(AR) + '-' + str(Re)
        path = run + '/constant/polyMesh/blockMeshDict'
        create_mesh_file(path, AR, Re)

        path = run + '/constant/transportProperties'
        create_properties_files(path, AR, Re)

    print ('Config generated.')
```

```python


def run_simulations(ARs, Res):
    for AR, Re in zip(ARs, Res):
        run = "Run" + str(AR) + '-' + str(Re)
        if not os.path.exists(run + '/log'):
            print(run + ' running now.')
            command = "hdiutil attach -quiet -mountpoint $HOME/OpenFOAM OpenFOAM.sparsebundle; "
            command += "sleep 1; "
            command += "source $HOME/OpenFOAM/OpenFOAM-2.3.0/etc/bashrc; "
            command += "cd " + run + "; "
            command += "blockMesh; "
            # command += "decomposePar; "
            # command += "mpirun -np 4 icoFoam -parallel > log; "
            # command += "reconstructPar; "
            # command += "foamCalc components U; "
            # command += "sample; "

            subprocess_cmd(command)
        print(run + ' complete.')

    print('Simulations complete.')


def main(ARs, Res):
    print('Running ARs ' + str(ARs) + ' with Res ' + str(Res) + '.')
    generate_folders(ARs, Res)
    update_dimensions(ARs, Res)
    run_simulations(ARs, Res)
    print('Done!')

if __name__ == "__main__":
    # Base case
    ARs = [ 2.0]
    Res = [100.0]
    main(ARs, Res)

    # Additional cases
    # ARs = [0.5,    0.5,   2.0,   5.0]
    # Res = [1.0, 2000.0, 100.0, 100.0]
    # main(ARs, Res)
```

Listing 1: Code to create solutions

```python
import numpy as np
import matplotlib
matplotlib.use('TkAgg')
import matplotlib.pyplot as plt

# Configure figures for production
WIDTH = 495.0  # the number latex spits out
FACTOR = 1.0   # the fraction of the width the figure should occupy
fig_width_pt = WIDTH * FACTOR

inches_per_pt = 1.0 / 72.27
golden_ratio = (np.sqrt(5) - 1.0) / 2.0      # because it looks good
fig_width_in = fig_width_pt * inches_per_pt  # figure width in inches
fig_height_in = fig_width_in * golden_ratio  # figure height in inches
fig_dims = [fig_width_in, fig_height_in]     # fig dims as a list

arr_befor = []
arr_after = []
for i in range(201):
    j = str(i)

    # Load data
    path = 'M=100/Run0.5-100.0/postProcessing/sets/' + j + '/'
    before = np.loadtxt(path + 'before_Ux_Uy.xy', delimiter=' ', unpack=True)
    after  = np.loadtxt(path + 'after_Ux_Uy.xy',  delimiter=' ', unpack=True)
```

```python

    Y, UX, UY = 0, 1, 2
    before_Y,  after_Y  = before[Y],  after[Y]
    before_UX, after_UX = before[UX], after[UX]
    # before_UY, after_UY = before[UY], after[UY]

    # Final flow should be linear
    linear = np.linspace(0, 1, len(before_UX))

    # Calculate RMS
    err_befor, err_after = before_UX - linear, after_UX - linear
    arr_befor.append(np.sqrt(np.mean(np.square(err_befor))))
    arr_after.append(np.sqrt(np.mean(np.square(err_after))))


# Plot
plt.figure(figsize=fig_dims)
plt.plot(arr_befor, 'b+', label='before')
plt.plot(arr_after, 'rx', label='after')
plt.yscale('log')
plt.legend()
plt.xlabel('Time (s)')
plt.ylabel('NRMS')

# plt.figure(figsize=fig_dims)
# plt.subplot(2, 1, 1)

# plt.plot(befor_error, before_Y, 'b+', label='before')
# plt.plot(after_error,  after_Y, 'rx', label='after')
# # plt.plot(middle_Y, middle_UX,  'g:', label='middle')
# plt.ylabel('y (m)')
# plt.xlabel('Error in Ux (m/s)')
# # plt.xlim([0, 1])
# plt.legend(loc='upper left')

# plt.subplot(2, 1, 2)
# plt.plot(before_Y, before_UY, 'b+',  label='before')
# plt.plot(after_Y,  after_UY,  'rx', label='after')
# # plt.plot(middle_Y, middle_UY,  'g:', label='middle')
# plt.xlabel('Y')
# plt.ylabel('Uy')
# plt.xlim([0, 1])
# plt.legend(loc='upper left')
plt.show()
```

Listing 2: Code to create plots